HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing

Michael Pellauer*, Michael Adler†, Michel Kinsy*, Angshuman Parashar†, Joel Emer*†

*Computation Structures Group
Computer Science and A.I. Lab
Massachusetts Institute of Technology
{pellauer, mkinsy, emer}@csail.mit.edu

†VSSAD Group
Intel Corporation
{michael.adler, angshuman.parashar,
joel.emer}@intel.com

***Abstract*—In this paper we present the HAsim FPGA-accelerated simulator. HAsim is able to model a shared-memory multicore system including detailed core pipelines, cache hierarchy, and on-chip network, using a single FPGA. We describe the scaling techniques that make this possible, including novel uses of time-multiplexing in the core pipeline and on-chip network. We compare our time-multiplexed approach to a direct implementation, and present a case study that motivates why high-detail simulations should continue to play a role in the architectural exploration process.**

***Index Terms*—Simulation, Modeling, On-Chip Networks, Field-Programmable Gate Arrays, FPGA**

I. Introduction

Gaining micro-architectural insight relies on the architect's ability to simulate the *target* system with a high degree of accuracy. Unfortunately, accuracy comes at the cost of simulator performance—the simulator must emulate more detailed hardware structures on every cycle, thus simulated cycles-per-second decreases. Naturally, there is a temptation to reduce the detail of the model in order to facilitate efficient simulation. Typical simulator abstractions include ignoring wrong-path instructions, or replacing core pipelines with abstract models. While such low-fidelity models can help greatly with initial pathfinding, the best way for computer architects to convince skeptical colleagues remains a cycle-by-cycle simulation of a realistic core pipeline, cache hierarchy, and on-chip network (OCN).

While parallelizing the simulator can recover some performance, parallel simulators have found their performance limited by communication between the cores on the OCN, and have been forced to reduce fidelity in the OCN in order to achieve reasonable parallelism [1], [2], [3]. In this paper we advocate an alternative approach—hosting the simulator on a reconfigurable logic platform. This is facilitated by an emerging class of products that allow a Field Programmable Gate Array (FPGA) to be added to a general-purpose computer via a fast link such as PCIe [4], HyperTransport [5], or Intel Front-Side Bus [6]. On an FPGA, adding detail to a model does not necessarily degrade performance. For example, adding a complex reorder buffer (ROB) to an existing core uses more of the FPGA's resources, but the ROB and the rest of the core will be simulated simultaneously during a tick of the FPGA's clock. Similarly, communication within an FPGA is fast, so there is great incentive to fit interacting structures like cores, caches, and OCN routers onto the same FPGA.

In this paper we present HAsim, a novel FPGA-accelerated simulator that is able to simulate a multicore with a high-detail pipeline, cache hierarchy, and detailed on-chip network *using a single FPGA*. HAsim is able to accomplish this via several contributions to efficient scaling that are detailed in this paper. First, we present a fine-grained time-multiplexing scheme that allows a single physical pipeline to act as a detailed timing-model for a multicore. Second, we extend the fine-grained multiplexing scheme to the on-chip network via a novel use of permutations. We generalize our technique to any possible OCN topology, including heterogeneous networks. We compare HAsim's time-multiplexing approach to a direct implementation on an FPGA. Finally, we use HAsim to study the degree that realism in the core model can affect OCN simulation results in a shared-memory multi-core, an argument for the continued value of high-detail simulation in the architectural exploration process.

This paper only considers a single FPGA accelerator. A complementary technique for scaling simulations is to partition the model across many FPGAs. However we do not consider this a limitation, as in order to maximize capacity of the multi-FPGA scenario we must first maximize utilization of an individual FPGA.

978-1-4244-9435-4/11/$26.00 ©2011 IEEE

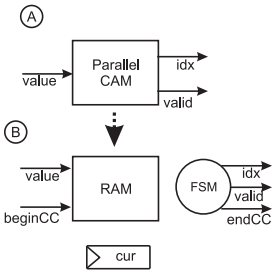


Fig. 1. (A) CAM Target (B) Simulating the CAM with a RAM and FSM over multiple FPGA cycles.

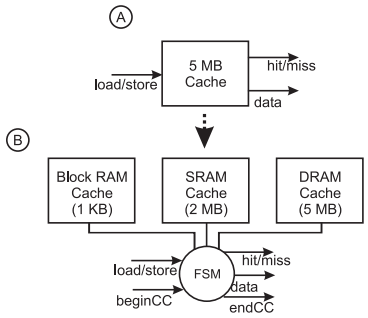


Fig. 2. (A) Large Cache Target (B) Simulating the cache using a memory hierarchy.

II. Techniques for Scaling FPGA-Accelerated Simulation

A. Background: FPGAs as Simulation Accelerators

Using FPGAs to accelerate processor simulation revolves around the realization that one tick of the FPGA clock does not have to correspond to one tick of the simulated machine's clock. The goal is not to configure the FPGA into the target hardware, but into a *performance model* that accurately tracks how many *model* clock cycles the operations in the target are supposed to take. This allows the model to simulate FPGA-inefficient structures using FPGA-efficient components, while using a separate mechanism to ensure their *simulated* timings match the target circuit.

For example, a Content-Addressable Memory (CAM) would be inefficient to implement directly on an FPGA, resulting in high area and a long critical path. However we can simulate a CAM using a single-ported Block RAM and an FSM that sequentially searches the RAM, as shown in Figure 1. The FSM may take more or fewer FPGA cycles to search the RAM, depending on occupation. However the model clock cycle is not incremented until the search is complete. Taking more or fewer FPGA cycles affects the rate of simulation, but does not affect the results. Thus the simulator architect is able to trade increased time for decreased utilization—if this tradeoff improves the FPGA clock rate and the FPGA-cycle-to-Model-cycle Ratio (FMR) remains favorable then this tradeoff is worth making. Detailed discussions of these techniques are given in [7], as well as Chiou [8],[7], Tan [9], and Chung [10].

Separating the model clock from the FPGA clock also allows the simulator to leverage the large amount of system memory in the host platform, even though the sizes and latencies may be radically different than those being simulated. In Figure 2 the simulator is run on a platform that has three levels of memory: on-FPGA Block RAM, on-board SRAM, and DRAM managed by the OS running on the host processor. The simulator wishes to use this hierarchy to simulate a 5 MB last-level cache. It can accomplish this by allocating space in the Block RAM, the SRAM, and host DRAM—essentially using 3 caches in place of a single large cache. To simulate an access of the target cache the FPGA first checks if the line is resident in the Block RAM. If it is, the simulator can quickly determine if the access hit or missed. Otherwise, it must access the SRAM or DRAM, and possibly add the response to the BRAM. In this case, in the rate of simulation will be slower, dependent on the distance of the memory where the line resides. But note that the level of physical memory accessed affects only the *rate* of simulation, and is orthogonal to whether or not the simulated 5MB cache hit or missed. To facilitate interfacing the simulator with the host system, HAsim uses the LEAP virtual platform [11], [12].

An FPGA-accelerated simulator is composed of many parallel modules, each of which can take an arbitrary number of FPGA cycles to simulate a model cycle. The problem now becomes connecting them together to form a consistent notion of model time. In HAsim this is done by representing the model using a *port-based* specification [7], as shown in Figure 3A. In such a specification the model is represented as a directed graph of modules connected by *ports*. In order to simulate a model cycle each module reads all of its input ports, computes local updates, and writes all of its output ports. If a module does not wish to transmit a message then it sends a special `NoMessage` value. Since each port has a message on it for every model cycle, the messages themselves can be thought of as *tokens* that enumerate the passage of model time. Port-based specifications predate FPGA implementation [13], but are a natural fit as they allow individual modules to make a local decision about whether to simulate the next cycle, without the need for global synchronization.

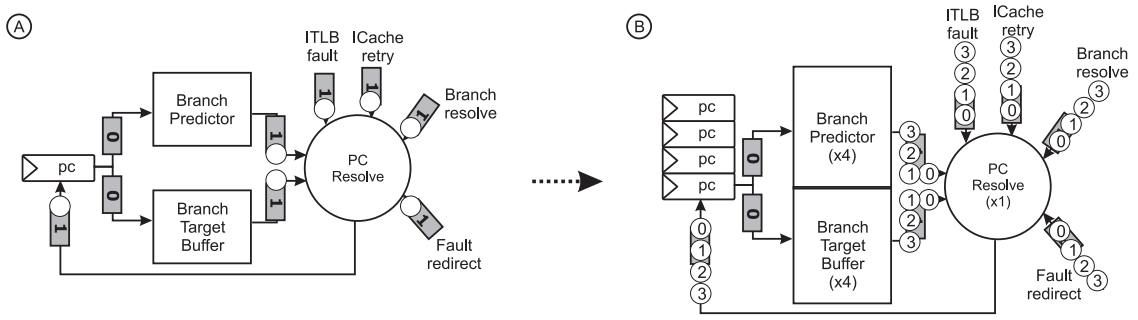


Fig. 3. (A) Port-based model of a processors' PC Resolve stage. (B) Time-multiplexing between 4 virtual instances.

	Functional Model	Timing Model	Time Multiplexed	Num Cores	Core Detail	OCN	Comments
Liberty [14]	N/A		No	16	*	No	*Uses hard PowerPCs on FPGA.
ProtoFlex [10]	FPGA	Software	Yes	16	*	No	*SMARTS-style functional/timing split.
UT-FAST [8], [15]	Software	FPGA	No	16	Yes	No	Software feeds trace to FPGA, which adds timing and may rollback software.
RAMP Gold [16]	FPGA	FPGA	Yes	64	No	No	Focuses on efficient simulation of cache models with abstract cores and no network.
HAsim	FPGA	FPGA	Yes	16	Yes	Yes	Model generalized cores, including out-of-order, superscalar.

Fig. 4. Comparison of FPGA-based processor simulators.

HAsim also employs a timing-directed approach, whereby the simulator is partitioned into a *functional* and *timing* model [17]. As in a traditional software simulator, the functional model is responsible for correct ISA-level execution, while the timing model adds micro-architecture specific timings such as branch predictions [18]. This technique is also employed by FPGA-accelerated simulators Protoflex [10], UT-FAST [8], and RAMP Gold [9]. In each case, the details of the partitioning schemes are different, as shown in Figure 4. The goal of the partitioning is to reduce the development effort associated with FPGAs: the functional model is written once, verified, optimized, and used across many different timing models.

B. Fine-Grained Time-Multiplexed Simulation

Separating the model clock from the FPGA clock can help with scaling specific structures within a target circuit, but experience has shown that it does not save enough space to allow duplicating high-detail cores, caches, and routers into a multicore configuration on a single FPGA.

Given this, *time-division multiplexing* is a technique that can help enable scaling our models to larger multicores. In such a scheme a single *physical* core is used to sequentially simulate several *virtual instances* that flow through the pipeline in sequence. Internal core state such as the program counter (PC) or register file (RF) is duplicated, but the combinational logic used to simulate each pipeline stage is not.[1] The disadvantage to time-multiplexing is that it can reduce simulation rate, as a single physical pipeline is being used sequentially to do the work of many.

The time-multiplexing approach was first used in the Protoflex simulator [10]. Protoflex multiplexes a functional model between 16 threads, but does not support any timing model on the FPGA. RAMP Gold [16] is another FPGA-accelerated simulator that uses a coarse-grained approach whereby a scheduler chooses a virtual instance to simulate, and performs the functional emulation of that instance without adding any timing model of the core. RAMP Gold does support timing models of caches, but does not currently support simulations of on-chip networks.

A contribution of HAsim is to extend previous multiplexing schemes to detailed timing models of core pipelines, while simultaneously minimizing any performance reduction from sequential time-multiplexing. HAsim accomplishes this by using the ports between modules to implement time-multiplexing: at simulator startup the ports are initialized with message tokens from each virtual instance, as shown in Figure 3B.

[1]This kind of multiplexing bears a resemblance to multi-threading in real microprocessors, but it is important to distinguish that this is a simulator technique, not a technique in the target architecture. The cores being multiplexed may or may not support multi-threading.

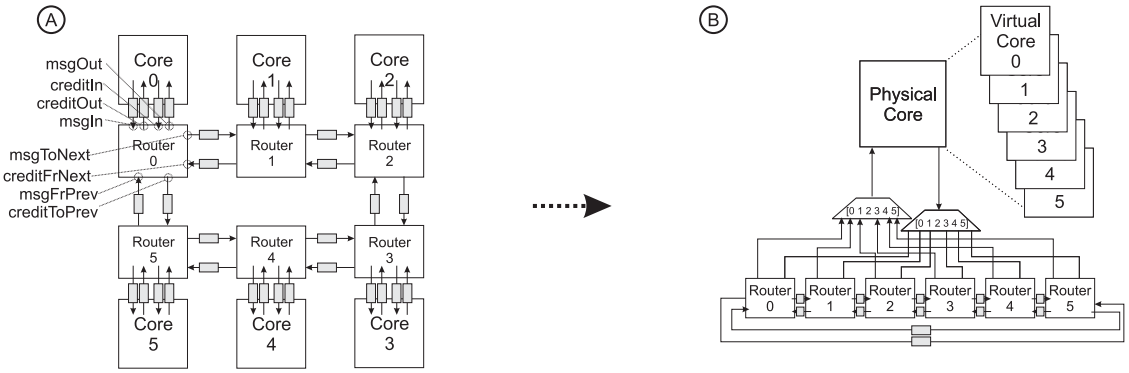


Fig. 5. (A) Target multicore with uni-directional ring network. (B) Multiplexed core connected to ring-network via sequential de-multiplexing.

In this scheme each stage of the core pipeline can be simulating a separate virtual core. For instance, the Fetch stage may be simulating Core 3 while the Decode stage simulates Core 2. Furthermore, modules that are implemented using multiple FPGA cycles per model cycle may themselves be pipelined.

This fine-grained time multiplexing minimizes impact on simulation rate by improving the utilization of the physical execution units. For example, if the CAM from Figure 1 were connected to the cache from Figure 2 then we would expect the rate of simulation to be limited by off-chip accesses in the cache, and during this time the CAM would mostly be idle. If this simulator were time-multiplexed, then the CAM module will not go idle until it has simulated all of the other virtual instances. Thus in many instances N-way time-multiplexing of a module does not slow the module's simulation rate by N. In fact, the simulation rate will not be affected at all until N grows beyond the current rate-limiting step. A study detailing the sub-linear slowdown of HAsim's scaling is presented in Section V-C.

Note that time-multiplexing scheme is possible only because the state of the different cores being simulated is independent. That is, the Register File of Core 0 cannot directly affect the Register File of Core 1. Only by going through the OCN can the various cores affect each other's simulation results. Because of this *cross-instance* communication, traditional time-multiplexing is insufficient for modeling the OCN—different techniques are needed that can take the interaction into account while still exploiting fine-grained parallelism.

III. Time-Multiplexed Simulation of On-Chip Networks via Permutations

A. First Approach: De-multiplexing

The previous section established that time-multiplexing the core works well because it improves both scaling and utilization. Now, the problem becomes attaching a single physical (time-multiplexed) core to an on-chip network. Consider the ring network shown in Figure 5A. Each router has 4 ports that communicate with the core/cache: `msgIn`, `creditIn`, `msgOut`, and `creditOut`. Additionally each router has 4 more ports that communicate with adjacent routers: `msgToNext, creditFromNext, msgFromPrev, creditToPrev`.

A baseline approach to simulating this target is to fully replicate the routers, and synthesize an on-chip network directly. The messages from the cores are then *sequentially de-multiplexed* and sent to the appropriate router. Each router can now simulate its next model cycle when data arrives. Responses are *re*-multiplexed and returned to the cores. This situation is shown in Figure 5B. In this figure and throughout the paper we represent sequential de-multiplexing by augmenting a de-multiplexor with a sequence denoting where each sequential arrival is to be sent. In this case the first arrival is sent to router 0, the second to router 1, and so on.

While this scheme is functionally correct, it presents many practical challenges. Most significantly, the physical core is now no longer adjacent to any particular router. Thus the FPGA synthesis tools are presented with the difficult problem of routing the de-multiplexed signals to the individual routers. Second, the routers themselves are under-utilized: at any given FPGA cycle only a small subset of routers are actively simulating the next model cycle—most are waiting for their corresponding virtual core to the produce data for a given model cycle. HAsim solves these problems by extending the time-multiplexing to the OCN routers themselves via a novel use of permutations.

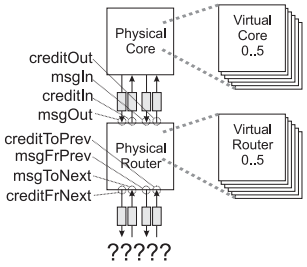


Fig. 6. Time-Multiplexing the ring is complicated by the cross-router ports/dependencies.

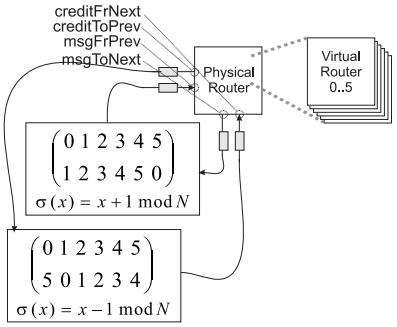


Fig. 7. Connecting the credit ports to each other, and the message ports to each other, and applying permutations to the messages.

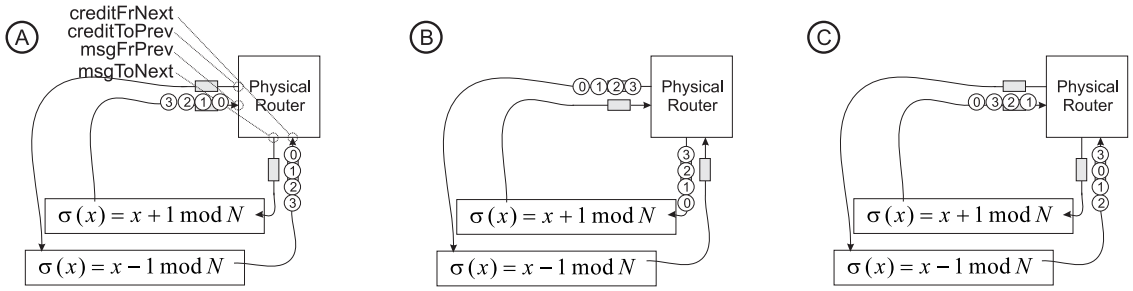


Fig. 8. Simulating a model cycle for ring network via permutations.

B. Time-Multiplexed Ring Network via Permutation

If we wish to time-multiplex the ring, observe that the simulation of router n is complicated by the communication from routers $n-1$ and $n+1$. As shown in Figure 6, it is the ports that cross between routers that present a challenge to time-multiplexing, as they express the fact that the differing virtual instances' behaviors are not independent. How can we ensure that each *cross-virtual instance* port's messages are transmitted to the correct destination?

The key insight, as shown in Figure 7, is that we can connect these ports to themselves. That is, the output from `msgToNext` is fed into `msgFromPrev`, and `creditFromNext` produces `creditToPrev`. This makes sense intuitively: messages leaving one router are the input to the next router. However, simply making the connection is not sufficient: router n produces the message for router $n+1$, not for router n.

One way to solve this would be to store cross-router communication in a RAM. The index of the RAM to be read and written by each virtual index would be calculated by accessing an indirection table. This approach is similar to the way a single-threaded software simulator simulates an on-chip network. The disadvantage is that a random-access memory is overkill, as the accesses are actually following a static pattern determined by the topology.

HAsim's insight is that the communication pattern can be represented by a small permutation. For the `msg` port the output from router 0 is the input for router 1 (on the next model cycle), 1 is for 2, and so on to $N-1$, which is for 0. For the `credit` port 0 goes to $N-1$, 1 to 0, 2 to 1, and so on. The advantage of this approach is that these permutations can be represented using two queues: a main queue and a side buffer. A small FSM determines which queue will be enqueued to, and which queue will be dequeued from.

Formally, given N cores the permutation σ for the xth input of each port is as follows:

- $\sigma_{msg}(x) = x + 1 \bmod N$
- $\sigma_{credit}(x) = x - 1 \bmod N$

In this paper we will express the permutations as shown in Figure 7: a concrete table showing that the output for core 0 is sent to core 1, and so on, until core 5's output is sent to core 0. This table is then supplemented with a generalized formula that scales the permutation to any number of routers.

Given these permutations, Figure 8 shows a complete example of simulating a model cycle in the ring network. In 8A the messages are in their initial configuration. The router simulates the next model cycle, consuming

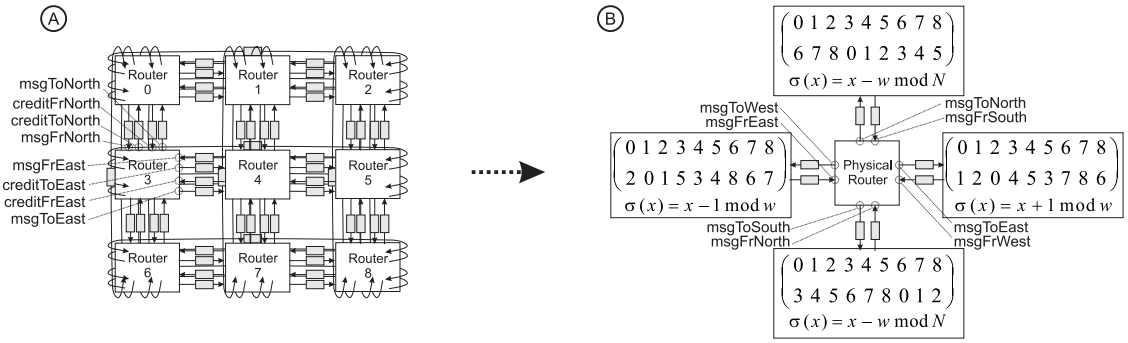


Fig. 9. Time-multiplexing a torus network. Cores/caches are not pictured. Credit ports are omitted as they use the same permutations.

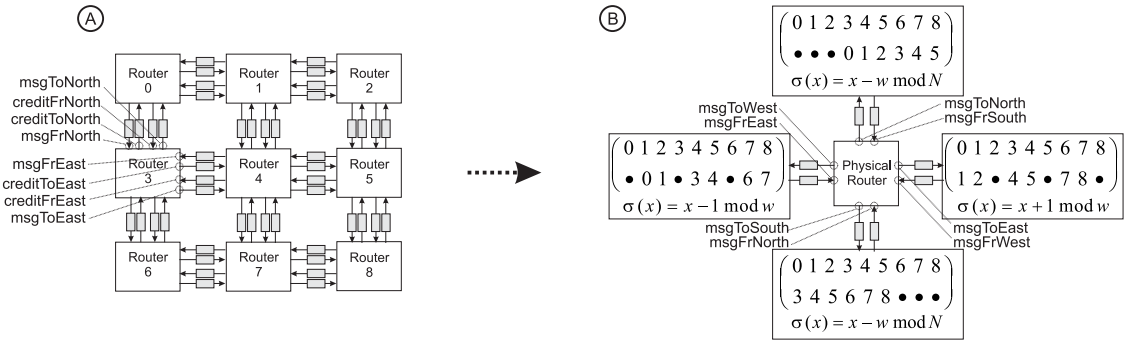


Fig. 10. Simulating a grid network using the same permutations as the torus and sending `NoMessage` messages on non-existent edges.

N inputs and producing N new outputs, resulting in the state shown in 8B. After the permutation is applied we can confirm that the resulting configuration in 8C is correct: on the next model cycle router 0 will receive the message from router 3, and the credit from 1. Router 1 will receive the message from router 0, and credit from router 2, and so on. Although we present this execution as happening in three separate phases, on the FPGA we can overlap the execution.

C. Time-Multiplexed Torus/Grid

Let us extend the permutation technique to another topology, the 2D torus shown in Figure 9A. Here each router has ports going to/from 4 directions: `msgFromNorth`, `msgFromEast`, `msgFromSouth`, `msgFromWest` and so on, as well as ports/to from the local core. As shown in Figure 9B the `msgToEast` port is connected to the `msgFromWest` port and so on, as expected. However, compared to the ring network the permutation is different to reflect the width of the torus. In order to simulate the cores in numeric order, the permutation for the East/West ports for a network of width w is:

- $\sigma_{msgFromEast}(x) = x + 1 \bmod w$
- $\sigma_{msgFromWest}(x) = x - 1 \bmod w$

Similarly the permutation for the North/South port must take into account the *width* of the network (not the height):

- $\sigma_{msgFromNorth}(x) = x + w \bmod N$
- $\sigma_{msgFromSouth}(x) = x - w \bmod N$

Note that these permutations mean that the output from router 0 will be sent to routers $\sigma_{msgFromNorth}(0) = 3$, $\sigma_{msgFromEast}(0) = 1$, $\sigma_{msgFromSouth}(0) = 6$, and $\sigma_{msgFromWest}(0) = 2$. Similarly router 0 will receive messages from $\sigma_{msgFromNorth}(6) = 0$, $\sigma_{msgFromEast}(2) = 0$, $\sigma_{msgFromSouth}(3) = 0$, $\sigma_{msgFromWest}(1) = 0$, corresponding exactly to the original target.

Once we have a torus model it is straightforward to alter this model to simulate a grid topology such as the one shown in Figure 10. We will not do this by altering the permutations or physical ports of our network, but rather by just altering the routing tables to send `NoMessage` (Section II-A) along the links that do not exist in the grid network. For instance, router 0, in the Northwest corner, will only send `NoMessage` West or North. If other routers obey similar rules then it will only receive `NoMessage` from those directions as well.

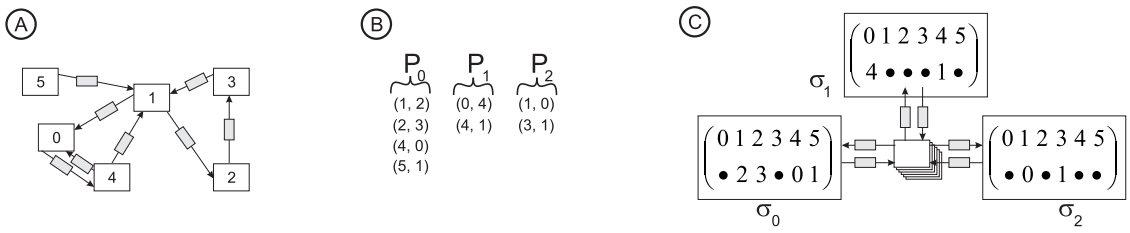


Fig. 11. Building permutations for an arbitrary network.

The permutations given in this section assume that the first processor that should be simulated (core 0) is located in the upper left-hand corner of the topology. If the architect desired a different simulation ordering they could accomplish this by changing the permutation — analogous to a sequential software simulator of a grid changing the order of indexing in a for-loop.

IV. Generalizing the Permutation Technique

The permutations described earlier correspond to picking the simulation order of the routers in the network and properly routing the data between them, similar to how a sequential software simulator cycles through nodes in sequence. It is always possible to create a sequential simulator for any valid OCN topology. In this section we demonstrate that it is similarly always possible to construct a set of permutations to allow any valid topology to be time-multiplexed.

A. Permutations for Arbitrary Topologies

Assume that the target OCN has been expressed as a port-based model: a digraph $G = (M, P)$ where M is the modules in the system and P is the ports connecting them. Label the modules M with a valid simulation ordering $[0, 1, .., n]$ such that 0 is the first node simulated and n is the last. Note that if the graph contains zero-latency ports then not all simulation orderings will be valid. However if the graph represents valid hardware then there is guaranteed to exist at least one valid simulation ordering.

Once the simulation ordering is picked we must combine the ports into as few time-multiplexed ports as possible. To do this we divide the edges P into the minimum number of sets $P_0, P_1..P_m$ such that each set P_m obeys the following properties:

- $\forall\{s, d\} \in P_m, \neg\exists\{s', d'\} \in P_m.s = s'$
- $\forall\{s, d\} \in P_m, \neg\exists\{s', d'\} \in P_m.d = d'$

In other words, no two ports in any given set can share the same source, or share the same destination. Each set P_m corresponds to a permutation that we must construct in our time-multiplexed model. Ensuring that no source or destination appears twice ensures that we will construct a valid permutation. We construct permutations $\sigma_{0..n} : M \rightarrow M$ using the following rule:

- $\forall\{s, d\} \in P_m, \sigma_m(s) = d$

The remaining range of σ_m represent "don't-care" values and so may be chosen in any way that creates a valid permutation. (It is possible that certain permutations will be cheaper to implement on an FPGA than others.)

Finally, each permutation should be associated with a port of the physical module. This module can be time-multiplexed using standard techniques (Section II-B), with one additional restriction: the time-multiplexed module should ensure that `NoMessages` are sent on port m for undefined values in the range of σ_m. This represents the fact that these output ports do not exist for a particular virtual instance. The torus/grid discussion in Section III-C is an example of this phenomenon.

Figure 11 shows an example applying this process to an arbitrary, irregular topology. First a desired simulation order is selected (11A). The ports are arranged into three sets (11B), the fewest possible for this example. These sets then form the basis of permutations (11C). The don't-care values of the permutations can be can be resolved in any way that creates a legal permutation. The router is time-multiplexed across 6 virtual instances, and the virtual instances are arranged to send `NoMessage` values on non-existent ports. For example, instance 0 will send `NoMessages` on two of the output ports, as the original router 0 only had one output port.

The meaning of undefined values in the permutations can clearly be seen when we apply the technique to a star network topology (Figure 12). The resulting time-multiplexed network has the same number of physical ports as the grid network, but the permutations themselves are different. Each leaf node only contains a subset of nodes of the hub, and thus will send `NoMessage` on ports that do not exist for them. Given this, the undefined values in the permutations can be filled in using straightforward modular arithmetic.

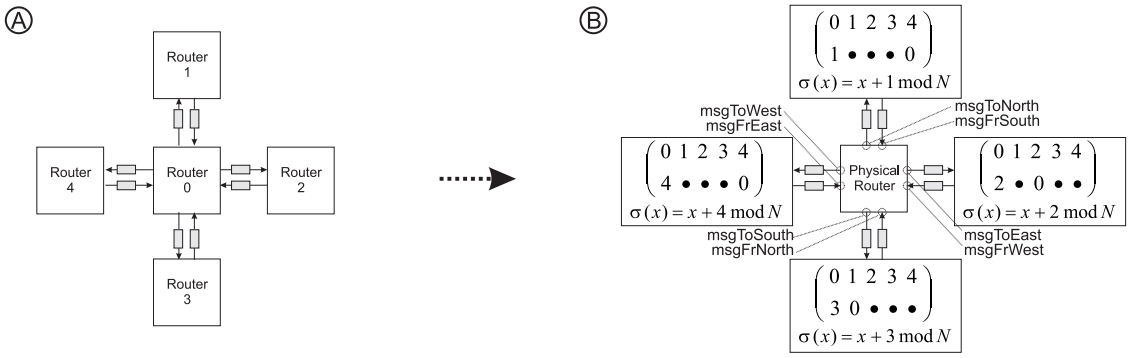


Fig. 12. Multiplexing a star topology results in many undefined values representing non-existent ports.

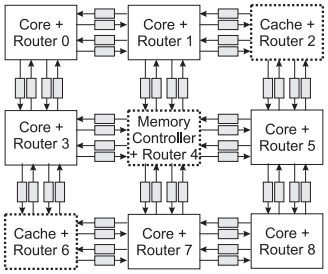


Fig. 13. A heterogeneous grid, where routers connect to different types of nodes.

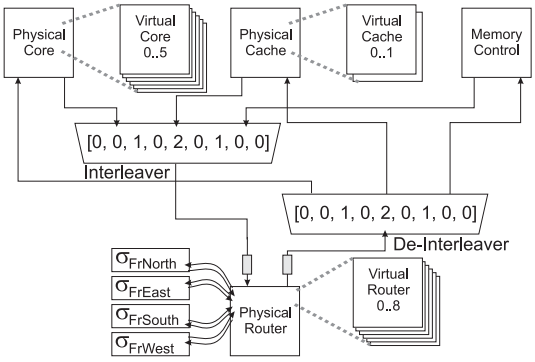


Fig. 14. Time-multiplexing the heterogeneous network via interleaving.

B. Heterogeneous Network Topologies

Thus far we have presented OCNs where all of the routers are connected to homogeneous cores. This has kept the examples pedagogically clear, but is unrealistic. Architects often wish to study multicores such as those shown in Figure 13, a 3x3 grid that contains a memory controller, 2 last-level caches, and 6 cores. The cores and caches will be simulated using time-multiplexing. How then can we connect them to our permutation-based grid? The answer is to sequentially multiplex the streams together, pass them to the time-multiplexed OCN, and de-multiplex the responses (Figure 14). Unlike the original de-multiplexing approach presented in Section III-A this imposes no difficult routing problem on the synthesis tools, as the modules being connected are time-multiplexed physical cores. A key advantage of this technique is that it requires no changes to the individual modules—they can be time-multiplexed independently using established techniques.

This same technique allows for efficient time-multiplexing of *indirect* network topologies such as butterflies, omitted for space considerations.

V. Assessment

A. Time-Multiplexing versus Direct Implementation

In this section we compare HAsim's time-multiplexed approach to Heracles [19], a traditional direct implementation of a shared-memory multicore processor on an FPGA. Heracles aims to enable research into routing schemes by allowing realistic on-chip-network routers to be paired with caches and cores, and arranged into arbitrary topologies. Heracles emphasizes parameterization in an effort to fit in many different existing FPGA platforms. A comparison of a typical Heracles implementation and a typical HAsim model is shown in Figure 15.

We synthesized both configurations using Xilinx ISE 11.5, targeting a Nallatech ACP accelerator [6], which connects a Xilinx Virtex 5 LX330T FPGA to a host-computer via Intel's Front-Side Bus protocol. The resulting FPGA characteristics are shown in Figure 16. Heracles is specifically made for efficiency, but the FPGA synthesis tools still have a problem scaling a complete system with core, cache, and router. This is because duplicating Heracles' caches exceeds the FPGA's Block

	Heracles	HAsim
Core		
ISA	32-Bit MIPS	64-Bit Alpha
Multiply/Divide	Software	Hardware
Floating Point	Software	Hardware
Pipeline Stages	7	9
Bypassing	Full	Full
Branch policy	Stall	Predict/Rollback
Outstanding memory requests	1	16
Address Translation	None	Translation Buffers
Store Buffer	None	4-entry
Level 1 I/D Caches		
Associativity	Direct	Direct
Size	16KB	16 KB
Outstanding Misses	1	16
Level 2 Cache		
Size	None	256 KB
Associativity	None	4-way
Outstanding Misses	None	16
On-Chip Network		
Topology	Grid	Grid
Routing Policy	X-Y DO Wormhole	X-Y DO Wormhole
Virtual Channels	2	2
Buffers per channel	4	4

Fig. 15. Component features of Heracles and HAsim.

	Registers	Lookup Tables	BlockRAM
Heracles			
2x2	44,512 (21%)	33,555 (16%)	328 (101%)
3x3	65,602 (31%)	59,394 (28%)	738 (227%)
4x4	DNF	DNF	DNF
HAsim (16-way multiplexed)			
4x4	120,213 (57%)	165,454 (79%)	88 (27%)

Fig. 16. Scaling a direct implementation versus the multiplexing approach.

RAM capacity. The synthesis tool was able to complete even in the presence of overmapping for the 2x2 and 3x3 configurations, but ran out of memory for the 4x4 case. We estimate that cache sizes would have to be reduced by a factor of 16 in order to successfully fit onto this FPGA.

In contrast, despite HAsim's significantly increased level of detail, we are easily able to fit a 4x4 multicore with L1 and L2 caches onto the same FPGA. This is due to four factors discussed earlier: First, separating the model clock from the FPGA clock allows efficient use of FPGA resources (Section II-A). Second, use of off-chip memory allows large memory structures like caches to be modeled using few on-FPGA Block RAM (Section II-A). Third, using a partitioned simulator allows HAsim to reduce the detail necessary in the timing model (Section II-A): it is well-known that timing models of caches need to store tags and status bits, but not the actual data. Most significantly, the HAsim 4x4 model is actually a single physical core, single cache, and single router that has been time-multiplexed 16 ways (Section III).

HAsim is an example of a space-time tradeoff. These techniques allow us to fit much more detail onto a single FPGA, paying for scaling by reducing simulation rate. Since at most one virtual instance can complete the physical pipeline per FPGA cycle, it takes a minimum of 16 FPGA cycles to simulate one model cycle. As the FPGA is clocked at 50 MHz, this gives HAsim a peak performance of $50/16 = 3.125$ MHz, multiple orders of magnitude faster than software-only industry models that are comparable levels of detail [8], [13].

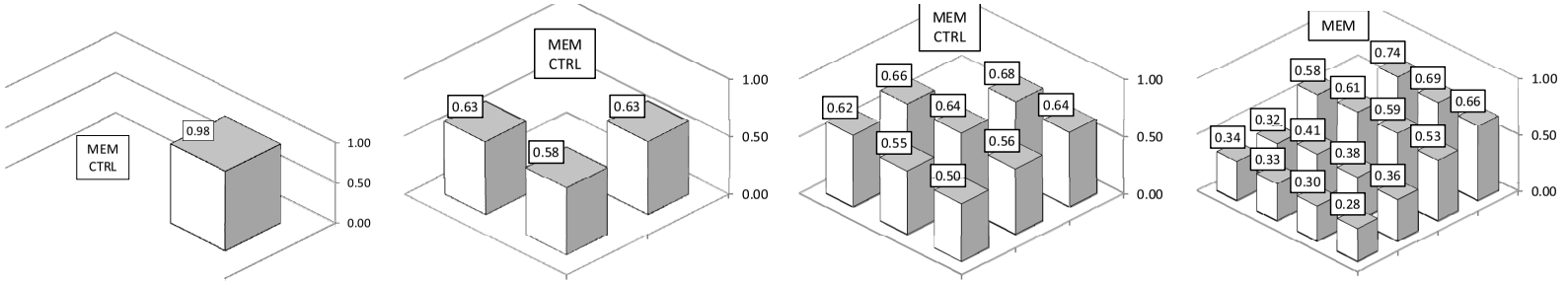
B. Case Study: Effect of Core Detail on OCN Simulation

It is not uncommon for architects who wish to study an OCN topology to reduce the level of detail in the core pipeline for the sake of efficient simulation. In such a situation the architect is hoping that the ability to run an increased variety of benchmarks will offset the increased margin of error of each run. It our hope that FPGA-accelerated simulators will present an alternative to reducing fidelity. This idea is particularly appealing if the FPGA means that the extra detail has minimal impact on simulation rate.

In order to evaluate the impact core fidelity can have on both simulation results and simulation rate, we modeled 2 multicore systems that differed only in the core pipelines. The first is a 1-IPC "magic" core running Alpha ISA that stalls on cache misses, similar to an architectural model. The magic core will never have more than one instruction in flight, and thus never produce more than one simultaneous cache miss. The second is the 9-stage pipeline described in Figure 15. This core does not reflect any particular existing architecture, but rather is representative of the general result of adding a higher-level of detail to the simulator.

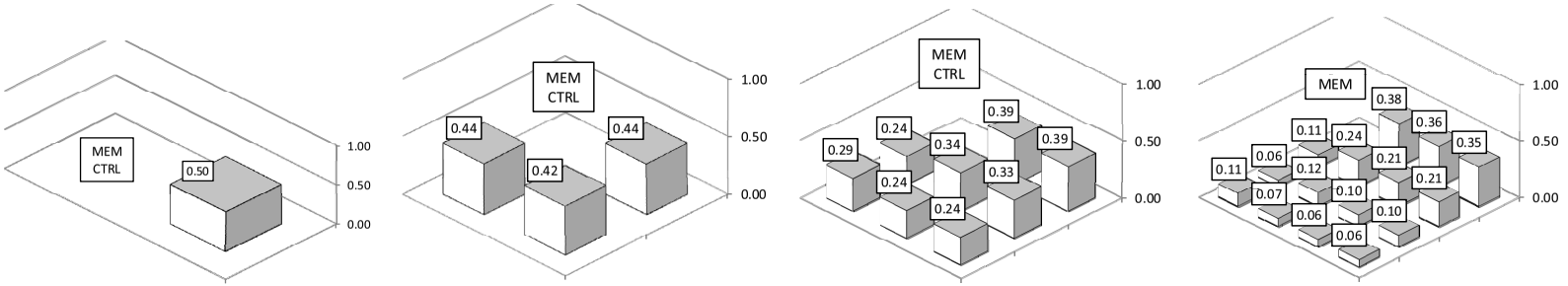
Each core was then connected to the cache hierarchy described in Figure 15 and arranged into 4 different grid configurations: 1x1, 2x2, 3x3, and 4x4. In each case one of the nodes was occupied by the memory controller, so the 4x4 configuration consisted of 15 core/cache pairs and 1 memory controller.

It is well-known that adding more cores to a shared-memory multicore can degrade the average IPC of each individual core, as contention on the OCN increases. This phenomenon represents a typical concern that an architect would like to characterize for a proposed OCN topology. We used HAsim to characterize the reported IPC of the individual cores running a variety of integer benchmarks, ranging from microkernels like Towers of Hanoi and vector-vector multiplication, to SPEC 2000 benchmarks gzip, mcf, and bzip2.



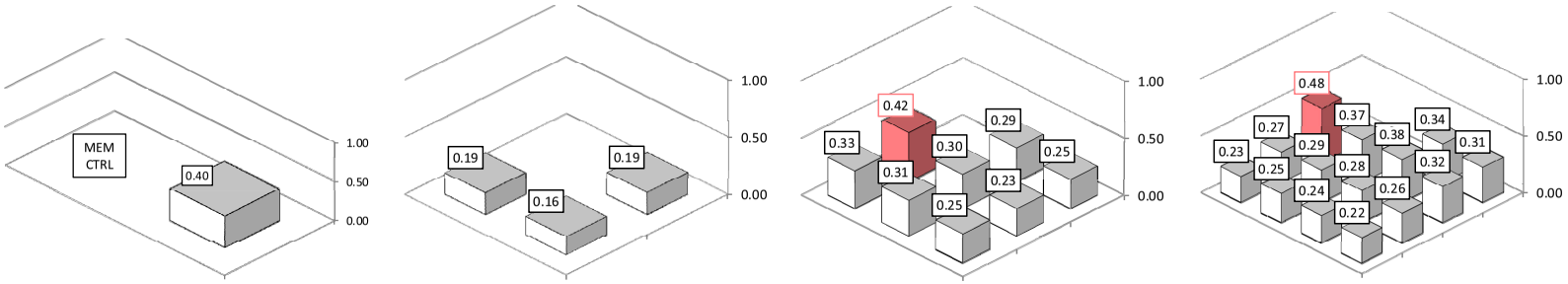
(a) 1x1 (b) 2x2 (c) 3x3 (d) 4x4

Fig. 17. Per-Core IPC: Magic Core Grids



(a) 1x1 (b) 2x2 (c) 3x3 (d) 4x4

Fig. 18. Per-Core IPC: Detailed Core Grids



(a) 1x1 (b) 2x2 (c) 3x3 (d) 4x4

Fig. 19. Absolute Difference in Reported IPC

The results are given in Figures 17-19. They demonstrate that the reported IPC of a particular core varies 0.16-0.48 between the two models. The most variation was shown by core (1,0) in the 4x4 model—the core directly south of the memory controller. This is because in the detailed model the cores south and east of this core generate more OCN traffic, due to simultaneous outstanding misses. The dimension-order routing scheme overwhelms core (1,0)'s ability to serve its local traffic. In the undetailed model the reduced contention allows (1,0) to sufficiently warm up its caches to run without network accesses. An architect studying the detailed model might conclude to move the memory controller, or institute a different routing policy—insights that might be missed when using the magic core.

All in all, these results indicate that high-detail simulation will remain a useful tool in the computer architect's toolbox.

C. Scaling of Simulation Rate

Now let us examine how HAsim's simulation rate scales as we add cores to the system. The time-multiplexing scheme means that simulating N processors has a best-case overall FPGA-cyle-to-Model-cycle Ratio (FMR) of N, with a best-case *per-core* FMR of 1.

As a baseline, a single-core model of our processor takes an average of 19.7 FPGA cycles to simulate a model cycle across a range of SPEC benchmarks. At first glance this seems to indicate that simulating N cores will reduce the FMR to $N * 19.7$. (FMR would scale linearly with the number of cores.) However, as noted in Section II-B, HAsim's fine-grained multiplexing at the port granularity means that the modules themselves are implemented in a pipelined fashion. This pipelining can lower the impact of time-multiplexing. In the best-case scenario the FMR of 19.7 would mean that we could simulate 19 virtual cores without impacting FMR at all, as we could finish the simulation of a core per FPGA cycle.

Unfortunately the situation is not so simple. Adding more virtual cores to the system impacts the per-core FMR of individual cores. This is because:

- Virtual cores increase cache pressure on the on-chip BRAM used to model the caches (Section II-A). This can reduce the FMR of the cores (though note that it has no impact on the simulation results themselves).
- The round-robin nature of the multiplexing scheme described in Section II-B means that when a particular virtual instance stalls for an off-chip access, the amount of work the rest of the system can perform is limited. For example, if we are simulating a 4-core system and Core 0 has an off-chip access then we can only simulate Core 1, 2, and 3 before we are back to 0 and cannot proceed.

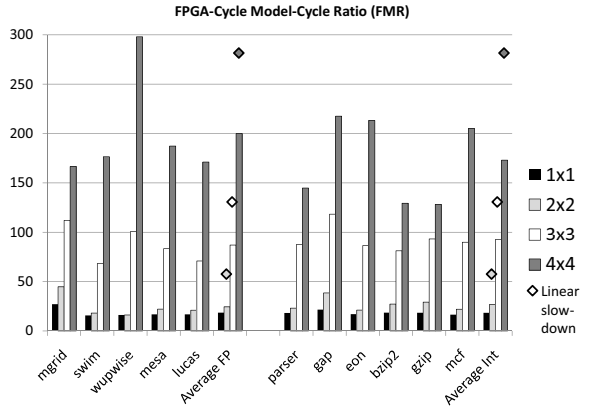


Fig. 20. Impact on FMR of scaling inorder core to multicore. The diamonds represent linear slowdown compared to the FMR of a single core.

	Min	Max	Average
FMR			
Overall	16	218	80
Per-Core	5	27	11
Simulation Rate			
Overall	160 KHz	3.2 MHz	625 KHz
Per-Core	1.84 MHz	9.5 MHz	4.54 MHz

Fig. 21. Comparing overall simulation rate to per-core rates.

Thus in the worst-case simulation rate could actually scale *worse* than linearly with the number of cores. To test this phenomenon we used the time-multiplexed inorder core scaling between 1x1 and 4x4, as described in the previous section.

The results of this scaling are shown in Figure 20. There are several interesting features of this graph that are worth exploring. First, note that when we scale from 1x1 to 2x2, the performance impact is quite minimal. In fact, in the case of the `wupwise` benchmark HAsim actually achieves the best-case scenario of not reducing FMR at all. This is because `wupwise` has a small working set that exerts very little cache pressure. On average the additional cache pressure slows the 2x2 simulation by 46% over the baseline. On average, this is significantly better than linear a slowdown of 300%, which is indicated by the diamonds on the graph. The fine-grained pipelining offsets the increased cache pressure, but not completely.

As we scale to 8 and 16 cores the increased cache pressure begins to have a greater impact. Although on aggregate we are still scaling better than linear slow-

down, the difference is clearly reduced. One interesting case is `wupwise`, which goes from having the best 2x2 simulator performance to having the worst at 4x4. It seems that once this benchmark's working set no longer fits in the on-chip cache the impact is quite extreme.

A breakdown of per-core FMR and simulation rate is given in Figure 21. It demonstrates that although the fastest simulator runs at 3.2 MHz, the average is 625 KHz. However, this rate is because we are simulating so many cores. The per-core simulation rate averages 4.54 MHz, peaking at 9.5 MHz in the best case.

As simulation rates are almost entirely limited by off-chip accesses, current research is focused on improving hit rates in the host memory hierarchy, either by an improved cache algorithms, or using a hardware platform with larger on-board DRAMs, or providing faster access to host memory. An alternative approach would be to loosen the round-robin multiplexing in order to keep the FPGA busy longer when off-chip accesses occur. Currently, no scheme is known that results in better performance at an acceptable hardware cost.

VI. Conclusion

Time-multiplexed simulation of detailed multicores using FPGAs represents a new tool in the architect's toolchest of simulation techniques. By trading space-savings for sequentialized simulation, it allows the possibility to free up substantial FPGA area. This critically limited resource can then be utilized to increase fidelity without negatively impacting simulation rate.

Alternatively, a natural extension of the techniques presented in this paper is to store the state of the virtual instances off-chip. Careful orchestration of memory accesses should be able to bury much of this latency and keep the physical pipeline busy. Currently we are aiming to use the techniques discussed here to model a thousand-node on-chip network using only a single time-multiplexed FPGA.

Acknowledgments

The authors would like to acknowledge the help and feedback of our collaborators in the RAMP project: Arvind, Derek Chiou, James Hoe, Krste Asanovic, John Wawrzynek. Other people who have contributed code to HAsim include Muralidaran Vijayaraghavan, Kermin E Fleming, Nirav Dave, Martha Mercaldi, Nikhil Patil, Abhishek Bhattacharjee, Guanyi Sun, and Tao Wang.

References

[1] J. Chen, M. Annavaram, and M. Dubois, "Slacksim: a platform for parallel simulations of cmps on cmps," *SIGMETRICS Performance Evaluation Review*, vol. 37, no. 2, pp. 77–78, 2009.

[2] J. Miller, H. Kasture, G. Kurian, C. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, January 2010.

[3] M. Lis, K. S. Shim, M. H. Cho, P. Ren, O. Khan, and S. Devadas, "DARSIM: a parallel cycle-level NoC simulator," in *Sixth Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, June 2010.

[4] HiTech Global Design and Distribution, LLC. `http://www.hitechglobal.com`, 2009.

[5] DRC Computer Corp. `http://www.drccomputer.com`, 2009.

[6] Nallatech, Inc. `http://www.nallatech.com`, 2009.

[7] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-ports: An efficient abstraction for cycle-accurate performance models on fpgas," in *Proceedings of the Sixteenth International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2008.

[8] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies FAST: Fast, full-system, cycle-accurate simulators," in *MICRO*, 2007.

[9] Z. Tan, A. Waterman, H. Cook, K. A. S. Bird, and D. Patterson, "A Case for FAME: FPGA Architecture Model Execution," in *Proceedings of the 37th International Symposium of Computer Architecture (ISCA)*, 2010.

[10] E. Chung, E. Nurvitadhi, J. H. K. Mai, and B. Falsafi, "Accelerating Architectural-level, Full-System Multiprocessor Simulations using FPGAs," in *FPGA '08: Proceedings Eleventh International Symposium on Field Programmable Gate Arrays*, 2008.

[11] A. Parashar, M. Adler, K. E. Fleming, M. Pellauer, and J. Emer, "LEAP: A virtual platform architecture for FPGAs," in *The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2010)*, December 2010.

[12] M. Adler, A. Parashar, J. Emer, K. E. Fleming, and M. Pellauer, "LEAP scratchpads: Automatic memory and cache management for reconfigurable logic," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2011.

[13] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim: A performance model framework," *Computer*, pp. 68–76, February 2002.

[14] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *The 12th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006.

[15] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The fast methodology for high-speed soc/computer simulation," in *International Conference on Computer-Aided Design (ICCAD)*, 2007.

[16] Z. Tan, A. Waterman, H. Cook, K. Asanovic, and D. Patterson, "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," in *Proceedings of the 47th Design Automation Conference (DAC)*, 2010.

[17] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick performance models quickly: Closely-coupled timing-directed simulation on FPGAs," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008.

[18] C. Mauer, M. Hill, and D. Wood, "Full-system timing-first simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30.1, pp. 108–116, 2002.

[19] M. Kinsy, "Heracles: Fully synthesizable parameterizable mips-based multicore system," Tech. Rep. MIT-CSAIL-TR-2010-058, MIT, 2010.